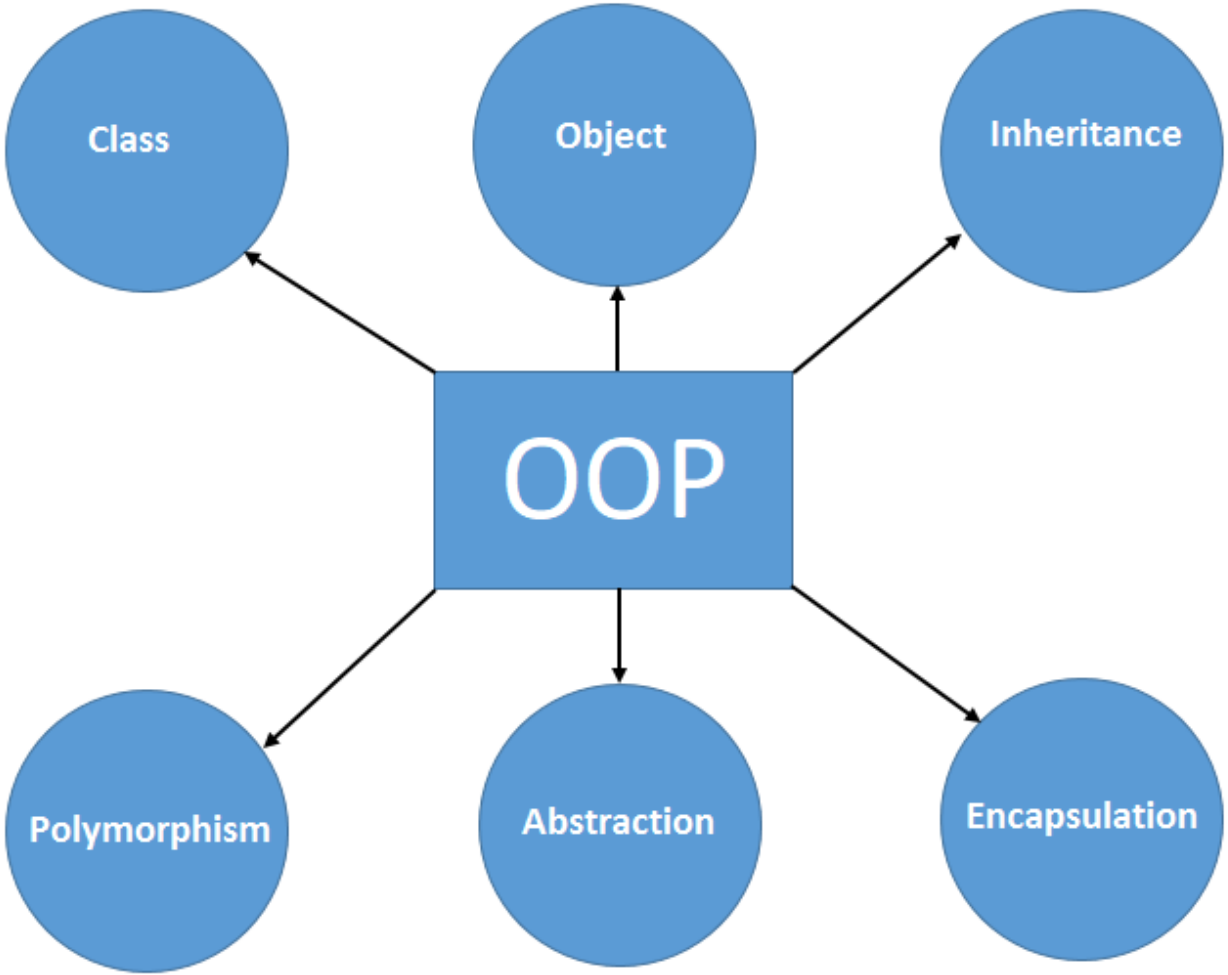
2. Advanced JavaScript Programming

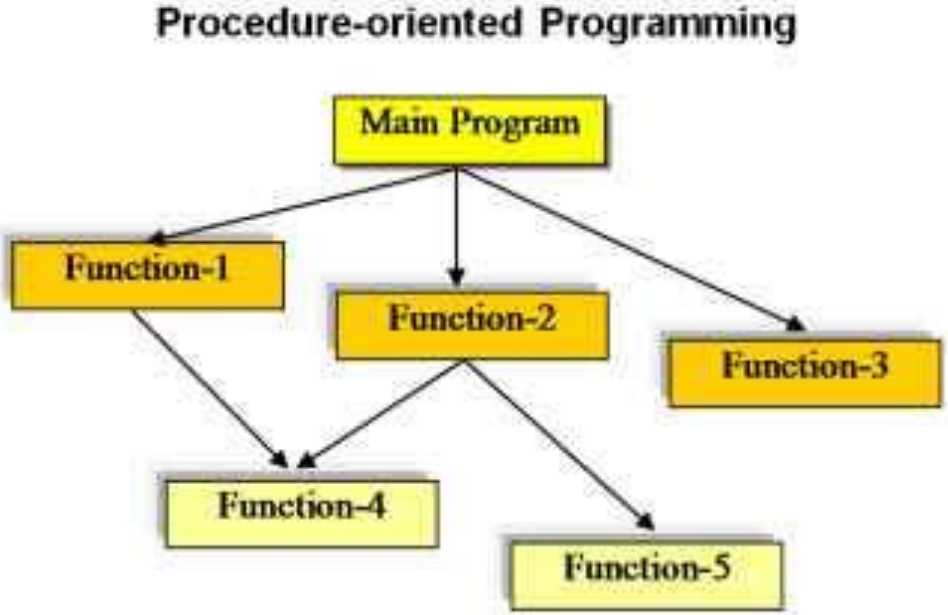
Adrian Adiaconitei

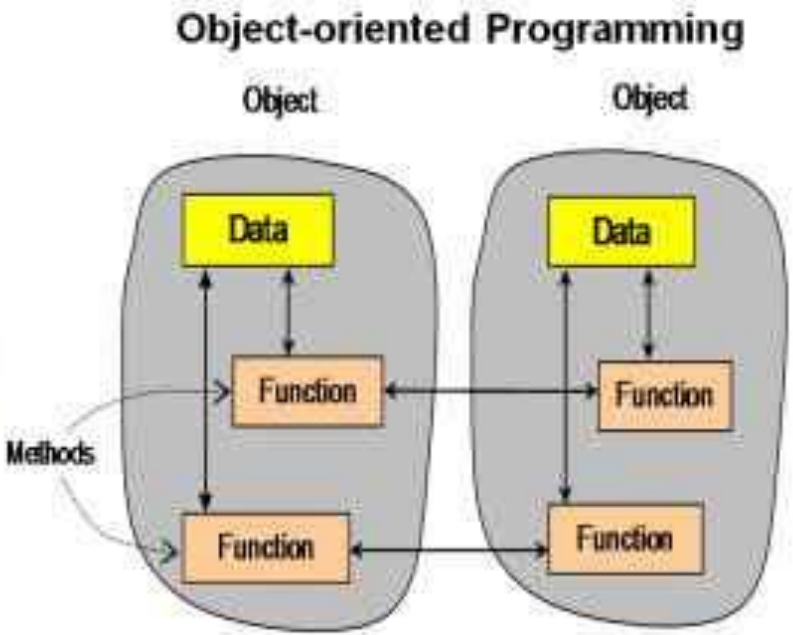
LINK Academy

Obiective

- ✓ Recapitulare
- ✓ OOP în JavaScript - refolosire de cod
  - ✓ Module
  - ✓ Call, Apply, Bind
  - ✓ Moștenirea cu ajutorul claselor și a prototipurilor
  - ✓ Încapsulare







JavaScript - Crearea obiectelor

1. Definim obiecte folosind metoda clasică: **Object literals**
2. Definim obiecte folosind metoda **Constructor function: return**
3. Definim obiecte folosind metoda **Constructor function: new**
4. Definim obiecte folosind metoda ***Object.assign*()**

https://jsfiddle.net/ShaikWasef/4xsden5w/

5. Definim obiecte folosind metoda **Object.create()**

https://jsfiddle.net/ShaikWasef/kcg08ypn/27/

6. Definim **class** – fabrică de obiecte

JavaScript - OOP

- ✓ **Clasa** este conceptul de baza în POO ce reunește *o colecție de obiecte , fabrică de obiecte*
- ✓ **Clasa este un șablon/schiță după care se generează obiectele, procedeu denumit: instanțierea clasei**
- ✓ O clasă va cuprinde definițiile datelor(atributelor / proprietăților) și operațiile (metodele) ce caracterizează obiectele de o anumită categorie.
- ✓ ***Obiectul*** este conceptul de bază în programarea orientată obiect(POO), care asociază datele împreună cu operațiile.
- ✓ **Obiectul** sau instanța este **un tip de date**, creat după un anumit model (clasă).
- ✓ **Datele** sunt informații de structură descrise de o mulțime de atribute ale obiectului, iar **operațiile** acționează asupra atributelor obiectului

JavaScript - OOP

JavaScript - OOP

- ✓ **Datele** definite într-o clasă se mai numesc atribute sau **proprietăți** / variabile, iar operaţiile se mai numesc **metode** sau funcții-membru / functiile.
- ✓ Proprietățile și metodele formează **membrii** unei clase.
- ✓ ***Definirea unei clase înseamnă crearea unui nou tip de date care apoi poate fi utilizat pentru declararea obiectelor de acest tip.***
- ✓ Fiecare clasă va avea identitate sau nume.
  - ✓ **NumeClasa**
  - ✓ **Proprietăți**
  - ✓ **Metode**

JavaScript - OOP

- ✓ Construirea obiectelor informatice pornind de la clase poartă numele de **instanțiere** sau exemplificare.
- ✓ ***Obiectul va fi o instanţă a unei clase.***
- ✓ Diferenţele dintre obiectele de aceeaşi clasă se materializează în diferenţe între valorile proprietăților.
- ✓ Pentru fiecare obiect este specificat tipul clasei din care provine.
- ✓ Pentru o clasă se pot crea mai multe instanțe ale acesteia.

JavaScript – OOP - Module

„ Autorii buni își împart cărțile în capitole și secțiuni; programatorii buni își împart programele în module.”

- ✓ **Mentenabilitate:** un modul este autonom. Actualizarea unui singur modul este mult mai ușoară atunci când modulul este decuplat de alte bucăți de cod.
- ✓ **Spațierea numelor:** modulele pot rezolva conflictele de nume. **Namespace** este paradigma de programare de a oferi un domeniu de aplicare identificatorilor (nume de tipuri, funcții, variabile etc.) pentru a preveni coliziunile între ei.
- ✓ **Reutilizare:** importăm și folosim mai ușor codul când este modularizat

JavaScript – OOP - Module

- ✓ Un modul este un fisier javascript. Utilizarea modulelor native JavaScript depinde de instrucțiunile:

1. Exportam: **export. (dupa nume și default)**
2. Importam: **import**
3. În fișierul HTML vom scrie <script **type="module"** src="main.js"></script>

https://html.spec.whatwg.org/multipage/scripting.html#the-script-element:javascript-mime-type

- ✓ Modulele funcționează întotdeauna în **“use strict”**
  - ✓ elimină unele erori JavaScript silențioase, si afiseaza acele erori.
  - ✓ codul în modul strict poate fi uneori făcut să ruleze mai rapid
  - ✓ facilitează scrierea JavaScript „securizat”.

Ap1.html

Ap2.html – conflict de nume

Ap3.html – import default

JavaScript – OOP - Module

Aplicația 1: Facem o aplicație ce va calcula aria, perimetru și desenează un triunghi, un cerc, un dreptunghi.
Pentru fiecare figură geometrică facem o clasă într-un fișier separat: modul.
Apoi exportăm/ importăm modulele în aplicația noastră.
Fiecare clasă conție metodele: deseneaza, calculeazaAria, calculeazaPerimetru

Soluție: app1

JavaScript – this

- ✓ Când este utilizat într-o funcție, cuvântul cheie **this** indică către un obiect la care este legat (obiectul curent).
- ✓ Există patru tipuri de legături:
  - ✓ legarea default
  - ✓ legarea implicită
  - ✓ legarea explicită
  - ✓ legarea apelului constructorului

JavaScript – this

- ✓ **call()** - vă ajută să înlocuiți valoarea **this** în interiorul unei funcții.
  func.call(thisObj, args1, args2, ...)

- ✓ **apply() -** este foarte asemănător cu call(). Singura diferență este că în **apply() accepta un array de argumente.** func.apply(thisObj, [args1, args2, ...]);

- ✓ **bind() -** creează o nouă funcție, când este invocată, are setările **this** la o valoare furnizată. Permite unui obiect să împrumute o metodă de la alt obiect fără a face o copie a acelei metode.
  func.bind(thisObj, arg1, arg2, ..., argN);

Aplicația2 :app2

JavaScript – OOP-Moștenire

- ✓ În JavaScript nu există moştenire clasică precum în alte limbaje de programare (C++, PHP, Java).
- ✓ Funcționalitatea se completează prin prototipuri. Din anumite puncte de vedere acest mecanism este mai puternic decât moștenirea.
- ✓ Toate obiectele JavaScript moștenesc metode și atribute de la un prototip.

Obiectele Date moștenesc Date.prototype. Obiectele Array moștenesc Array.prototype. Obiectele Person moștenesc Person.prototype.

- ✓ Object.prototype stă la baza ierarhiei tuturor protipurilor:
- ✓ Date, Array și Person moștenesc Object.prototype.

JavaScript – OOP- Moștenire

- ✓ Prin prototipuri putem adăuga metode și proprietăți noi unei clase existente.
- ✓ Pentru aceasta putem folosi proprietate prototype.

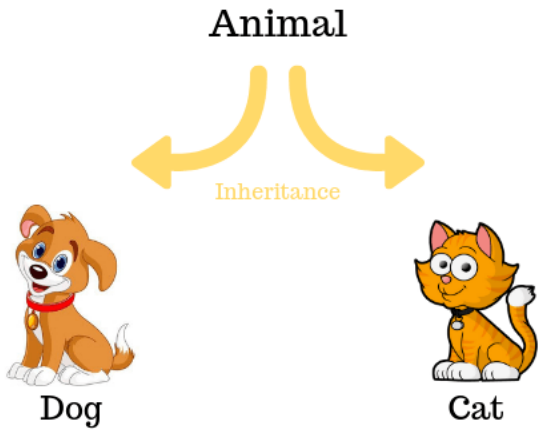
Ap4.html - moștenire prin prototipuri înainte de ES6

Adăugați clasei Array posibilitatea de eliminarea tuturor numerelor pare prin metoda removeEven, folosind proprietata prototype.

```
var arr = [1,2,3,4,5,6,7,8,9,10];
arr.removeEven();
console.log(arr);
```

JavaScript – OOP- Moștenire

- ✓ **Moştenirea** este procesul prin care un obiect poate să preia prototipul altui obiect. Acest lucru este important deoarece se admite conceptul de clasificare.



- ✓ Fără utilizarea claselor, fiecare obiect ar trebui definit explicitându-se toate caracteristicile sale.
- ✓ Mecanismul moştenirii este acela care face posibil ca un obiect să fie un exemplar specific al unui caz mai general.
- ✓ Scopul principal al moștenirii este: refolosirea de cod și evitarea codului duplicat

JavaScript – OOP- Moștenire

- ✓ cuvântul cheie **extends**, prin care acesta poate indica faptul ca o clasă este derivată dintr-una deja existenta.
- ✓ Clasa derivate(copil / subclasă) preia în acest fel, parțial sau total, atributele și metodele clasei originale(părinte / superclasă)
- ✓ cuântul cheie **super** din clasa copil apelează constructorul sau altă metodă din clasa părinte
- ✓ Deoarece **super**() inițializează **this**, trebuie să apelați **super**() înainte de a accesa **this**.
- ✓ Dacă în clasa copil nu este definit constructorul se apelează automat constructorul părinte

PHP OOP- Moștenire

```
class Carte{
        titlu;
        autor;
        toString(){
        console.log (`Cartea se numeste ${this.titlu} si este scrisa de ${this.autor}`);
        }
}
class Manual extends Carte{// Clasa derivate(copil / subclasa)
        nrbuc;
         getComanda(){
         console.log (`Cartea ${this.titlu} este comandata in ${this.nrbuc} exemplare`);
        }
}
```

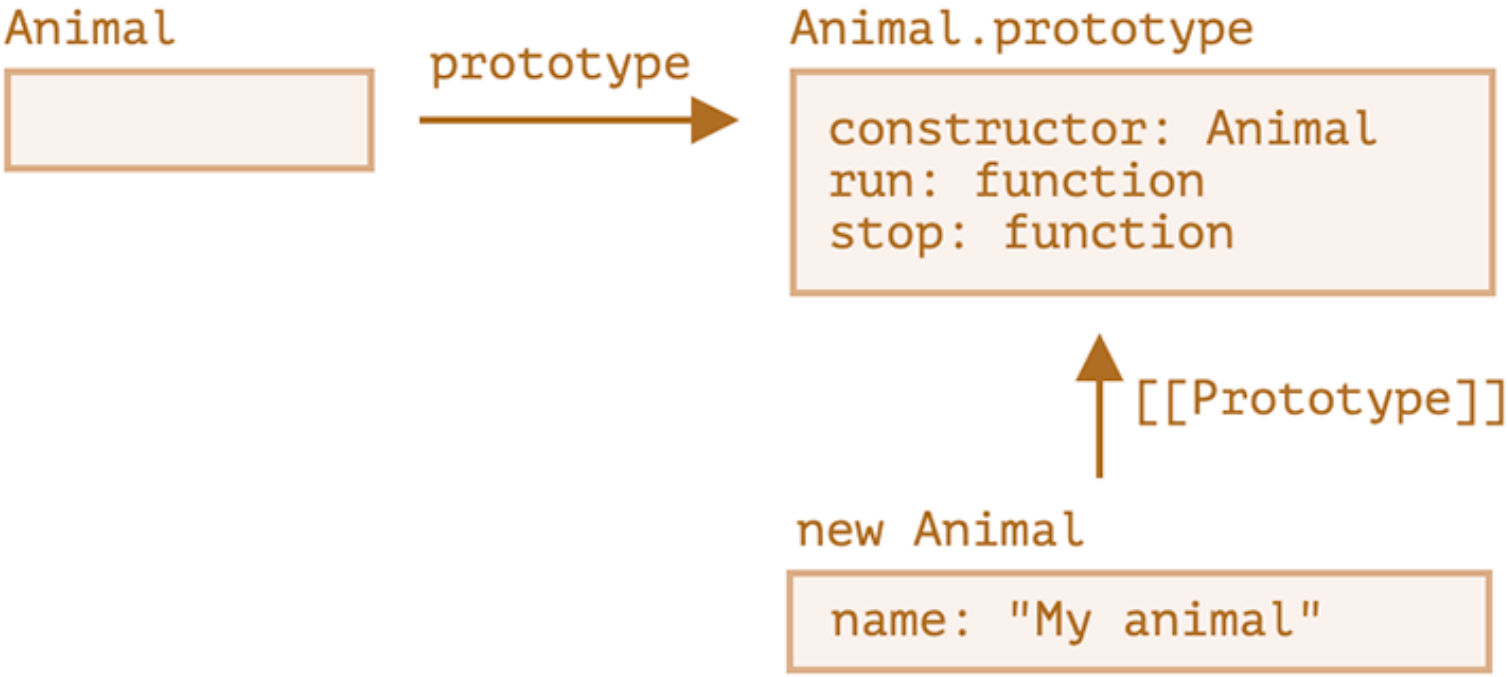

Ap5.html – folosind module

JavaScript – OOP- Moștenire

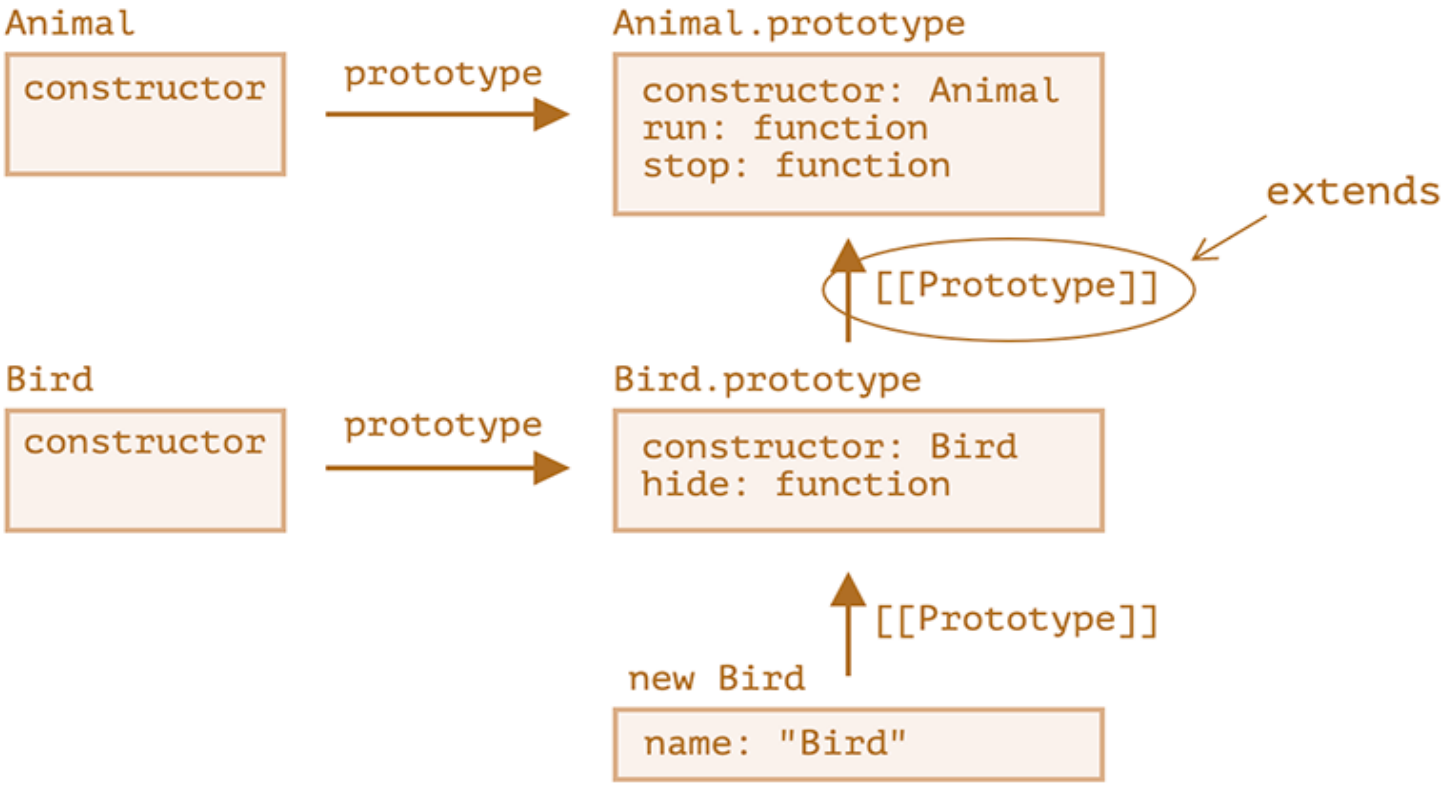
```
class Animal {
        constructor(legs) { this.legs = legs; }
        walk() { console.log('walking on ' + this.legs + ' legs’); }
}
class Bird extends Animal {
        constructor(legs) { super(legs); }
        fly() { console.log('flying'); }
}
let bird = new Bird(2);
bird.walk();
bird.fly();
```

Ap6.html- folosind module

JavaScript – OOP- Moștenire



JavaScript – OOP- Moștenire

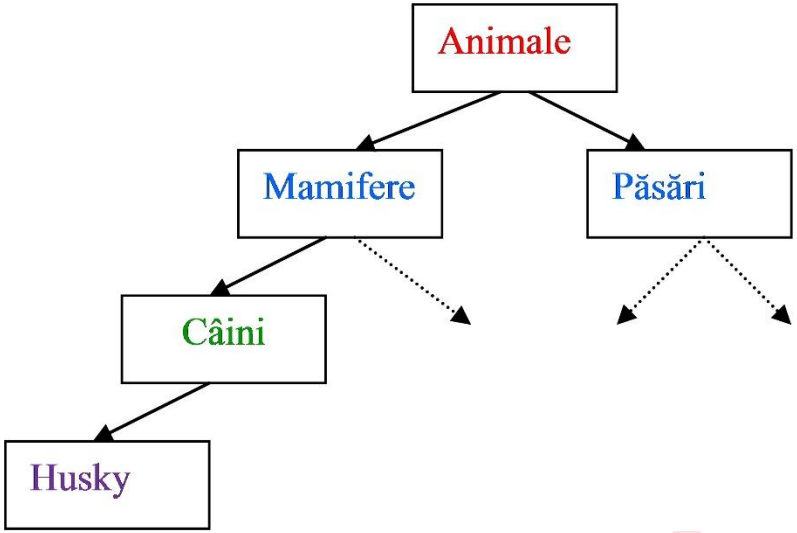


JavaScript – OOP- Moștenire

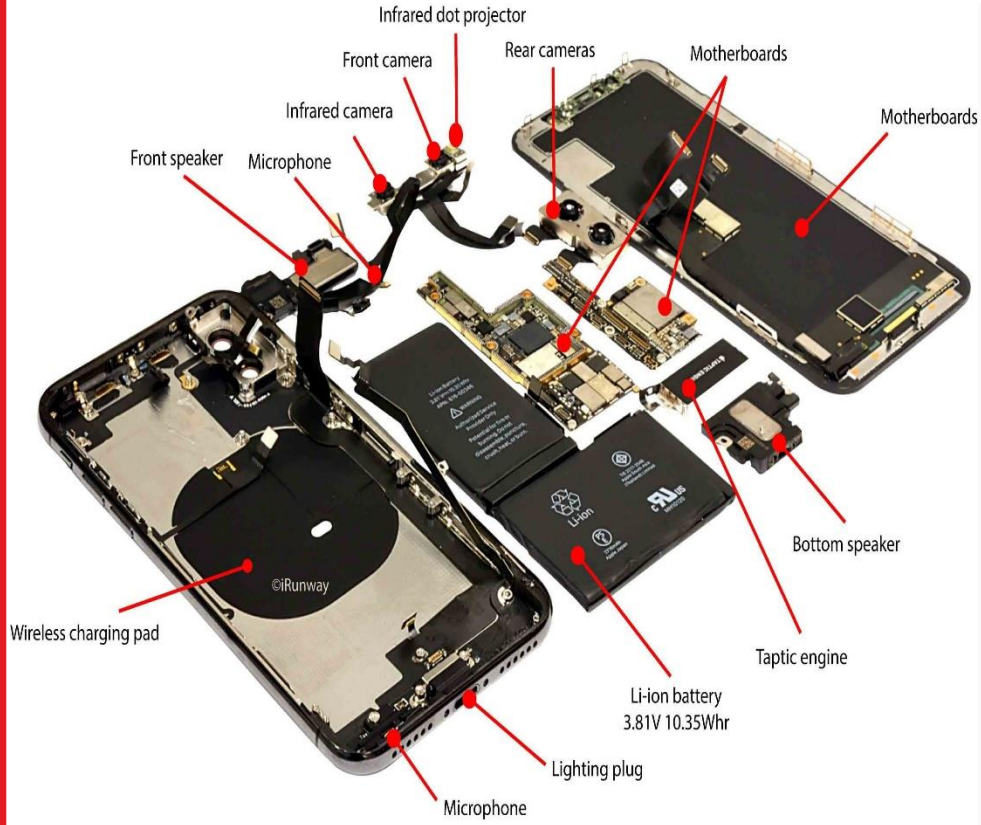
- ✓ **Overriding** - posibilitatea de a rescrie implementarea metodelor moștenite
- ✓ Atentie! Metoda rescrisă să aibă același nume și să nu aibă nivel de acces mai restrictiv decât în părinte!
- ✓ Lista de argumente poate însă diferi de cea din clasa părinte dacă au valori default (argumentele fără valori default sunt obligatorii)

JavaScript – OOP- Moștenire

- ✓ **Moştenirea** pe mai multe nivele



JavaScript – OOP– Încapsulare

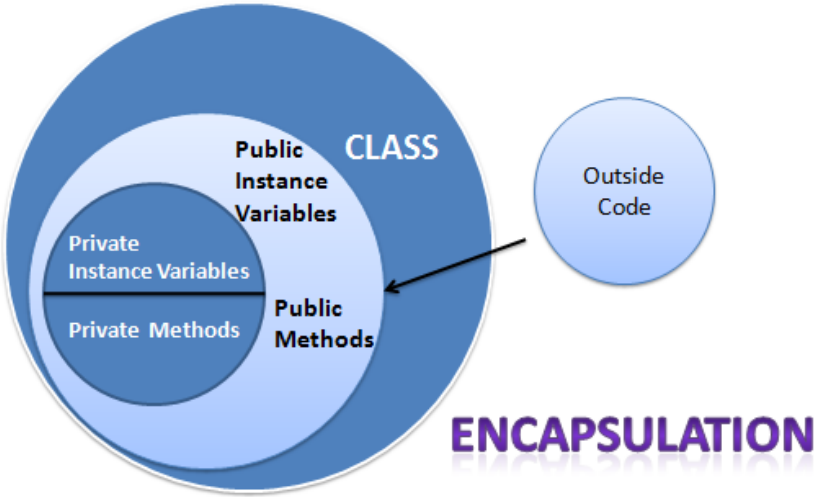


JavaScript – OOP– Încapsulare

- ✓ **Încapsulare** este procesul prin care ținem datele și metodele protejate de exterior. **Încapsularea** nu reprezintă altceva decât proprietatea claselor de a grupa sub aceeași structură datele și metodele aplicabile asupra datelor.
- ✓ Un **obiect** este o entitate logică ce încapsulează atât date cât și cod care manevrează aceste date. Într-un obiect o parte din cod și/sau date pot fi particulare acelui obiect şi inaccesibile

în afara sa. În acest fel, un obiect dispune de un nivel semnificativ de protecție care împiedică modificarea accidentală

Pentru a accesa datele se folosesc metode de tip: **setter** și **getter**



OOP– Încapsulare

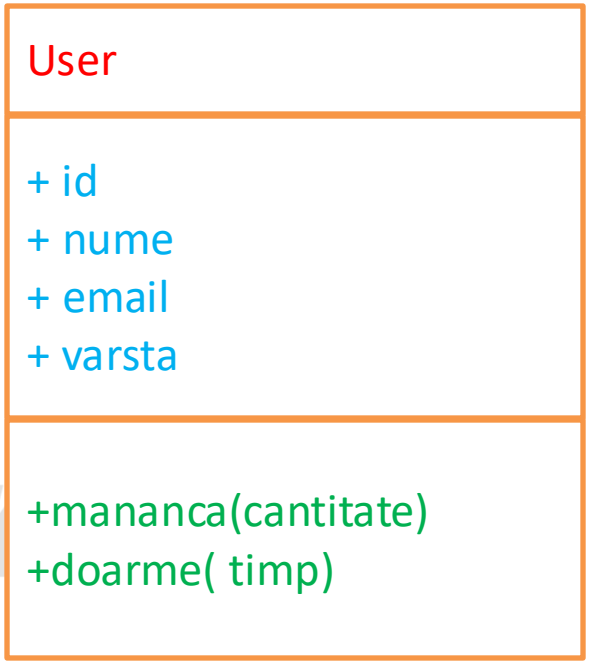
- ✓ Modificatorii de acces sunt **private, protected** și **public**.
- ✓ Se aplică membrilor unei clase (proprietați sau metode) pentru a determina vizibilitatea acestora.
- ✓ Membrii clasei au nivel de acces implicit **public**.
- ✓ un membru **public** este vizibil atât în interiorul metodelor clasei, cât și prin intermediul unei variabile obiect.
- ✓ un membru **protected** este vizibil in interiorul clasei în care a fost definit și în clasele derivate din aceasta
- ✓ un membru **private** nu este vizibil decât în interiorul metodelor din cadrul clasei în care a fost definit, fiind "ascuns" în interiorul obiectului.

JavaScript – OOP– Încapsulare

1. un membru **public** este vizibil atât în interiorul metodelor clasei, cât și prin intermediul unei variabile obiect.

Ap7.html

Definim o clasă User , ca modul, conform schemei UML cu toate proprietățile și metodele publice



JavaScript – OOP– Încapsulare

2. un membru **protected** este vizibil in interiorul clasei în care a fost definit și în clasele derivate din aceasta

Ap8.html

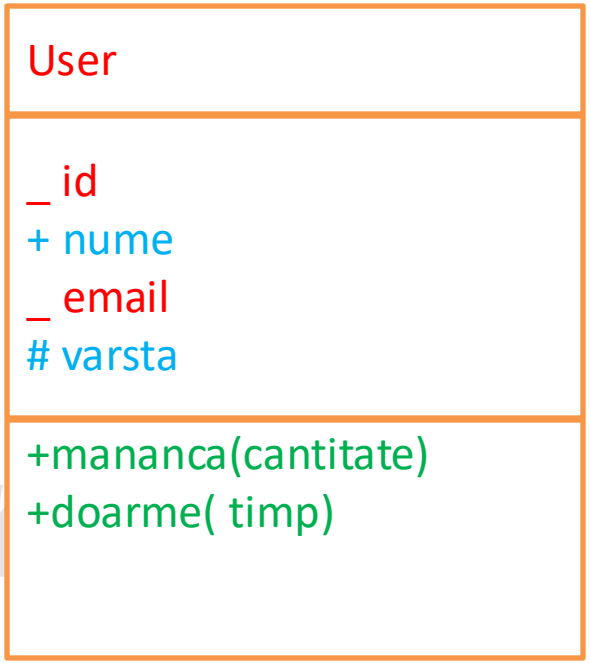
Modificam clasa User, ca modul, conform schemei UML cu proprietățile și metodele private folosind setter și getter

(# in javascript – private)

+ pentru proprietăți /metode public

_ pentru proprietăți /metode protected

pentru proprietăți /metode private ES2019/ES10



Aplicația 3: Definiți o clasa Animal cu urmatoarele:

✓ Prorpietăți:

Private familie (vertebrate, nevertebrate, domestice, salbatice)

Private mancare(ierbivor, carnivor, omnivor)

Public greutate

Public static nrPicioare

Private culoare

Const nrOchi

✓ Metode:

Public mananca(cantitate)

Public doarme(timp)

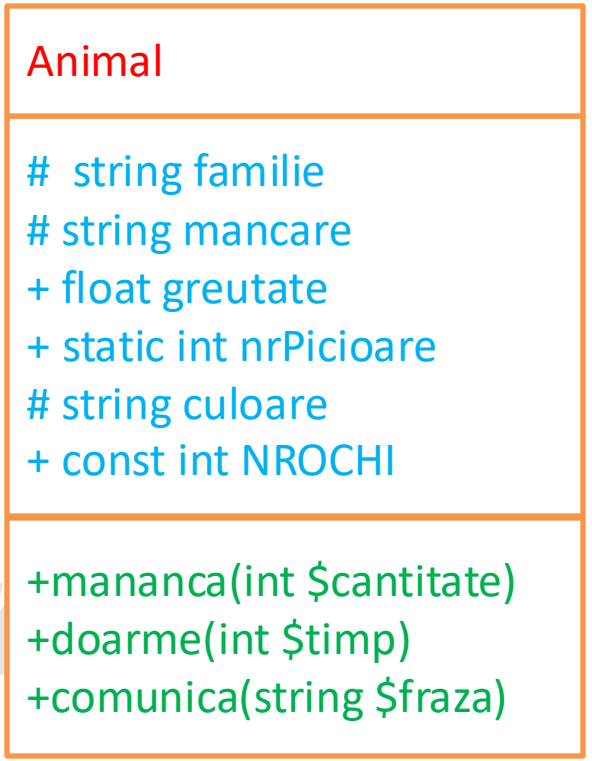
Public comunica(fraza)

Aplicația 3: Unified Modeling Language (UML) | Class Diagrama

+ pentru proprietăți /metode public

_ pentru proprietăți /metode protected

pentru proprietăți /metode private ES2019/ES10



✓ Pentru fiecare atribut al clasei definiți metode de tip **setter** și de tip **getter**

✓ Facem 2 clase copil Caine, Leu, pentru clasa Animal

Resurse

https://medium.com/dailyjs/using-import-aliases-in-javascript-a0b46237601c

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming#encapsulation

https://dev.to/bhagatparwinder/classes-in-js-public-private-and-protected-1lok

https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_uml_basic_notation.htm